Hackers Guide to Visual FoxPro 6.0

S2C4. Productive Debugging.

Productive Debugging

Error is a hardy plant: it flourisheth in every soil.
Martin Farquhar Tupper, *Proverbial Philosophy*

Our programs run correctly the first time, every time. Yeah, right—if you believe that one, we have a bridge in Brooklyn we'd like to sell you. We figure we spend at least as much time debugging our programs as we do writing them in the first place.

If your programs do run correctly the first time, every time, you can skip this part of the book. In fact, you should write a book and tell us how to do it. For the rest of you, we'll take a look at the kinds of errors that occur in programs, give you some ideas on how to root them out and show how VFP can aid your efforts.

"Whatever Can Go Wrong, Will"

I will not steep my speech in lies; the test of any man lies in action.
Pindar, *Olympian Odes IV*

Errors in programs fall into three broad categories. One of them is pretty easy to find, the second usually isn't too bad and the third is the one responsible for most of our gray hairs (at least a few of those have to be credited to our respective children).

The first group of errors, the easy ones, is the syntax errors. These are the ones that come up because you typed "REPORT FROM ..." or forgot the closing quote on a character string, and so on. These are easy because FoxPro will find them for you if you ask. (In fact, VFP 5 increased the strictness of syntax checking so even more of them than before can be found just for the asking.)

If you just start running the program, it crashes as soon as an error is found. Not bad, but you can do better. Use the COMPILE command or choose Compile from the Program menu and FoxPro checks the whole program for syntax errors and gives you an ERR file (with the same name as your program). One shot and you can get all of these. The Build option in the Project Manager gives you the same errors for all of the code within a project. We regularly select "Recompile All Files" as an easy "smoke test" for the project—just make sure to also check "Display Errors" to see the resulting ERR file.

The next group is a little more subtle. We call these "runtime errors" because they don't turn up until runtime. These errors result from comparing variables of different types, dividing by 0, passing parameters when none are expected, and so forth. They're things that FoxPro can't find until it actually runs the code, but then they're obvious. Again, your program crashes as soon as one of these turns up.

Because you have to tackle them one at a time and you have to actually execute a line of code to find an error in it, these are more time-consuming to deal with than syntax errors. But they're manageable. In VFP 5 and later, they're even more manageable because you can use assertions to test many of these things, so you can find them before VFP crashes. The task gets easier again in VFP 6 because the Coverage Profiler can help you figure out what's been tested, so you can test the rest.

The truly terrible, difficult errors are what we call "correctness errors." Somehow, even though you knew what the program was supposed to do and you carefully worked out the steps necessary to do it, it doesn't do what it should. These are the errors that try programmers' souls.

Tracking down correctness errors requires a planned, systematic, step-by-step attack. Many times, it also requires you to take a fresh perspective on the symptoms you're seeing—take a break or ask someone else to look at your code. (Tamar has solved countless correctness errors by talking them out with her non-programmer husband. Ted's favorite method is to try to explain the problem in a CompuServe message—invariably, two-thirds of the way through explaining it, the problem explains itself.)

Don't You Test Me, Young Man

None but the well-bred man knows how to confess a fault, or acknowledge himself in an error.
Benjamin Franklin, *Poor Richard's Almanac*, 1738

So now you can categorize your errors. So what? What you really want to know is how to get rid of them.

Let's start with a basic fact. You are the worst possible person to test your code. It's fine for you to track down syntax and runtime errors, but when it's time to see if it works, anyone else will do it better than you.

Why's that? Because you know how it's supposed to work. You're going to test the system the way it's meant to be used. No doubt after a little work, you'll get that part working just fine. But what about the way your users are really going to use the system? What happens when they push the wrong button? When they erase a critical file? When they enter the wrong data?

Understand we're not picking on you personally—we're just as bad at testing our own applications. The psychologists call it "confirmatory bias." Researchers find that a programmer is several times more likely to try to show that a program works than that it doesn't.

So how can we avoid the problem? The answer's obvious: Get someone else to test our code. That's what all the big companies do. They have whole testing departments whose job it is to break code. If you're not a big company, you could try what one of our friends used to do—he hired high school students to come in and break his code. We've been known to sit our spouses or kids down in front of an app and let them bang on it.

A second problem we've found is that clients are almost as bad at testing as we are. Not only do they have an investment in the success of the application, but very typically they hired you because they did not have the resources to develop the application—they won't find the resources to test it, either! The ideal situation is one in which the testing person's interest, motivation and job description is to find flaws in the program.

No matter who's testing the code, you need a structured approach to testing. Don't just sit down and start running the thing. Make a plan. Figure out what the inputs and outputs should look like (or should never look like!), then try it.

Here are some things you need to be sure to test (and it's easy to forget):

- Every single path through the program. You need to figure out all the different branches and make sure every single one gets tested. Change your system date if you have to, to test the end-of-decade reporting features or what happens on February 29, 2000.
- Bad inputs. What if the user enters a character string where a number is called for? What if he enters a negative invoice amount? What if she tries to run end-of-month processing on the 15th?
- Extreme values. What happens if the payment is due on January 1, 2001? What if someone's last name is "Schmidgruber-Foofnick-Schwartz"?
- Hitting the wrong key at the wrong time. We've seen applications that crash dead when you press ESC.

In larger projects, you should generate a set of test data right up front that handles all the various possibilities and use it for ongoing testing. It's easy to figure out the expected results once, then test for them repeatedly.

With large projects, it's also much more likely for a change in one place to cause problems in another. Plan for regression testing (making sure your working system hasn't re-exposed old buggy behavior due to changes) for these applications. Your test data really comes in handy here.

You need to test your error handler, too. The ERROR command makes it easy to check that it handles every case that can occur and does something sensible. When you design the error handler, keep in mind that every new version of FoxPro that's come along has introduced new error codes—be sure it's easy to add them.

Where the Bugs Are

Truth lies within a little and certain compass, but error is immense.
Henry St. John, Viscount Bolingbroke, *Reflections upon Exile*, 1716

Once you figure out that the program's broken, what next? How do you track down those nasty, insidious bugs that haunt your code?

FoxPro has always had some decent tools for the job, but starting in VFP 5 (that's getting to be a theme in this section, isn't it?), the task is much easier. We'll talk only about the Debugger introduced in that version. If you're using VFP 3, check out the *Hacker's Guide to Visual FoxPro 3.0* for suggestions on debugging using the older tools.

Assert Yourself

Let me assert my firm belief that the only thing we have to fear is fear itself.
Franklin D. Roosevelt

The way to start debugging is by keeping certain kinds of errors from happening in the first place. The ASSERT command lets you test, any time you want, whether any condition you want is met. Use assertions for any conditions that can be tested ahead of time and don't depend on user input or system conditions. If it can break at runtime even though it worked in testing, don't use an assertion. We use ASSERT the most to test parameters to ensure that they're the right type and contain appropriate data.

Give your assertions useful messages that help you hone right in on the problem. One trick is to begin the message with the name of the routine containing the assertion. A message like:

```
MyRoutine: Parameter "cInput" should be character.
```

is a lot more helpful than the default message:

```
Assertion failed on line 7 of procedure MyRoutine.
```

or, the worst, a custom message like:

```
Wrong parameter type.
```

Liberal use of assertions should let you get your code running faster and help to track down those nasty regression errors before they get to your clients.

Debugger De Better

Error is the contradiction of Truth. Error is a belief without understanding. Error is unreal because untrue. It is that which seemeth to be and is not. If error were true, its truth would be error, and we should have a self-evident absurdity—namely, erroneous truth. Thus we should continue to lose the standard of Truth.
Mary Baker Eddy, *Science and Health*, 1875

Before VFP 5, FoxPro could just barely be said to have a debugger. The Watch and Trace windows worked as advertised and gave you tools for seeing what was going on, but they were pretty limited. Fortunately, as so often happens with FoxPro, the development team heard our pleas and VFP 5 introduced "the new debugger," so called because it doesn't really have any other name.

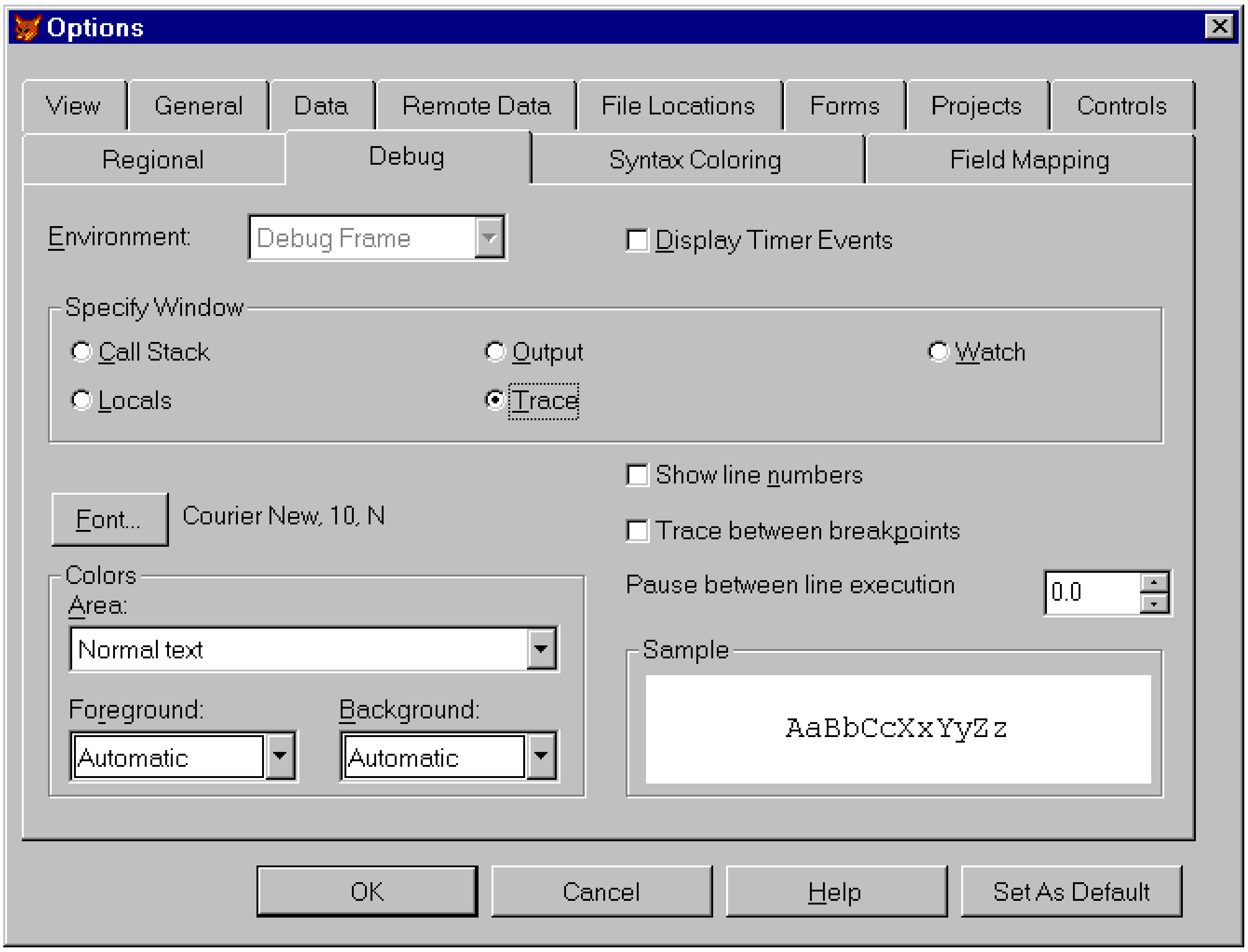
The debugger is composed of five windows and several other tools. It can run inside the VFP frame (meaning that the windows are contained in the main VFP window and are listed individually on the Tools menu) or in its own frame, with its own menu and its own entry on the taskbar. On the whole, we much prefer putting the debugger in its own frame. Then we can size and position both VFP and the debugger as we wish (someday we hope to have a monitor big enough to make them fit together nicely without compromises—perhaps our best hope is using two monitors side by side), minimize the debugger when we're not using it, and so forth. In addition, when the debugger has its own frame, tools like Event Tracking and Coverage Logging appear in the debugger's Tools menu. When the debugger lives in the FoxPro frame, those tools appear only on the debugging toolbar that opens when any of the debugger's windows are opened. We should point out that, even when in its own frame, the debugger is still not totally independent of VFP. It closes when VFP closes, and you can't get to it when you're in a dialog in VFP. (Well, actually, you can get to it by bringing it up from the taskbar, but you can't do anything there.)

The five main debugger windows are Trace, Watch, Locals, Output and Call Stack. Whichever frame you use, each of them is controlled individually.

You can open or close whichever windows are useful at the moment. Although we call them "windows," they're all actually toolbars and can be docked at will. In fact, these windows let you decide whether they can be docked. Right-click on any of them. If "Docking View" is checked, the window can be docked (by dragging or double-clicking). If "Docking View" is unchecked, double-clicking maximizes the window. Weird, but it's pretty cool to have that much control. It's interesting to note that Microsoft thinks debugging is so different from everything that goes on in the Windows platform that it can have unique window characteristics and nonstandard behavior. Or maybe this is the next interface paradigm coming down the road. We think not.

So what do these windows let you do? Trace and Watch are a lot like the old Trace and Debug windows, but with a lot more power. Locals saves on putting things into the Watch window—it lets you see all the variables that are in scope at the moment. Call Stack shows you the (nested) series of calls that got you to the current routine. Output, also known as Debug Output, holds the results of any DebugOut commands, as well as things you consciously send there, like events you're tracking. We'll take a look at each one, and point out some of the cooler things you can do.

Before that, though, a quick look at the Debug page of the Tools-Options dialog. This is one of the strangest dialog pages we've ever encountered. Near the middle, it has a set of option buttons representing the different debugger windows. Choosing a different button changes the dialog, showing options appropriate only to that window. Figure 2-1 shows the dialog when the Trace window is chosen. Figure 2-2 shows the dialog when the Output window is chosen. Each window also has its own font and color settings.


Options

View | General | Data | Remote Data | File Locations | Forms | Projects | Controls
Regional | Debug | Syntax Coloring | Field Mapping

Environment: Debug Frame

[ ] Display Timer Events

Specify Window
( ) Call Stack ( ) Output ( ) Watch
( ) Locals [✓ Trace]

[ ] Show line numbers

Font... Courier New, 10, N

[ ] Trace between breakpoints

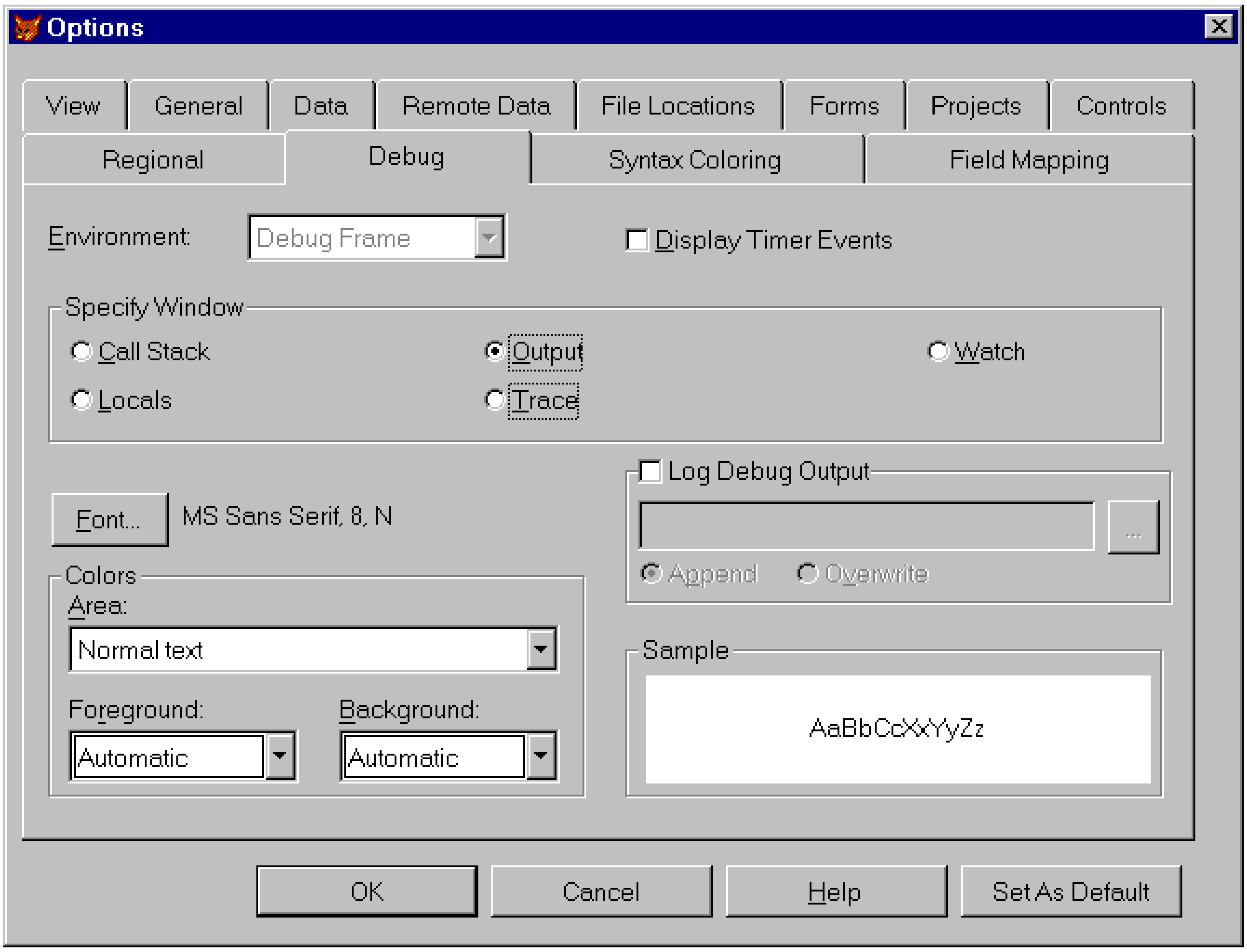
Pause between line execution 0.0

Colors
Area: Normal text
Foreground: Automatic
Background: Automatic

Sample
AaBbCcXxYyZz

OK | Cancel | Help | Set As Default

FIGURE 2-1: Tracing Paper—When the Trace window is chosen, you can specify whether to track between breaks and how long to wait between each line executed.


Options

View | General | Data | Remote Data | File Locations | Forms | Projects | Controls
Regional | Debug | Syntax Coloring | Field Mapping

Environment: Debug Frame

☐ Display Timer Events

Specify Window
Call Stack [✓ Output] Watch
Locals Trace

☐ Log Debug Output
[✓ Append] Overwrite

Font... MS Sans Serif, 8, N

Colors
Area: Normal text
Foreground: Automatic
Background: Automatic

Sample
AaBbCcXxYyZz

OK | Cancel | Help | Set As Default

Figure 2-2: Putt-Putt Output—With the Output window chosen, you can indicate whether to store any output sent to that window to a file, and if so, where.

One last general point: Since you're likely to be working on a number of programs and each probably has different debugging needs, you can save the debugger's current configuration and reload it later to pick up where you left off.

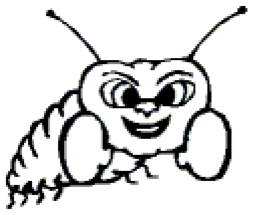
The Trace of My Tears

Well had the boding tremblers learned to trace
The day's disasters in his morning face.
Oliver Goldsmith, *The Deserted Village*

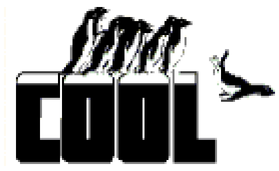
The Trace window shows you code and lets you go through it at whatever pace you want. You can open programs ahead of time or wait until a Suspend in the code or a breakpoint stops execution. Once a program is open in the Trace window, you can set a breakpoint on any executable line. When you run the program, execution stops just before running that line.

In an application, the Trace window (together with the Call Stack window) gives you quick access to any program or object in the execution chain. You can also open up others and set breakpoints by choosing Open from the Debugger menu or using the Trace window's context menu.

Several commands let you execute a little bit of the program while still retaining control. Step Into executes the next command, even if it calls another routine. Step Out executes the rest of the current routine, stopping when it returns to the calling routine. Step Over executes an entire routine without stepping through the individual lines within; if you use it when you're on a line that's not a call to a subroutine, it's the same as choosing Step Into. Run to Cursor lets you position the cursor in the code, then execute all the code up to the cursor position. All of these options have menu shortcuts, so you can use a single keystroke (or a keystroke combination) to move on. (If some of the keystrokes don't work for you, check your development environment. Even if the debugger is running in its own frame, On Key Labels you've defined intervene and prevent keystrokes from executing debugger commands.)



When you run the Debugger in the FoxPro frame, the F7 (Run to Cursor) and Shift+F7 (Step Out) keystrokes don't work. They're fine when the Debugger lives in its own frame.



The Set Next Statement choice on the Debug menu is a little gem. While stepping through code, you can position the cursor, and then choose this one to skip over some code or go back and execute some again. This is probably one of the most overlooked options in all of VFP. Thanks to our friend Christof Lange for pointing it out to us.

The Pause between line execution option on the Trace portion of the debugger options (see Figure 2-???) is pretty cool. Sometimes you want to step through a program, but the simple act of doing so changes the results. This setting to the rescue—it lets you slow the program down (not one of our usual goals, but handy for debugging). Set it to a speed where you can see each line execute without growing a beard in between. Note that the dialog simply changes the system variable _THROTTLE.

Trace between breakpoints is also controlled from the Tools-Options dialog, as well as from the Trace window's context menu. It can make a big speed difference when you're trying to get to a particular trouble spot. It determines whether all code is echoed in the Trace window on the way to a breakpoint. If you turn it off, Trace stays the same until you reach a breakpoint, then the code there is updated—this can be a lot faster.

Opening a form in the Trace window is a pain. You can't just use File-Open as you can with a PRG. However, the new Breakpoints dialog makes it easy to set a breakpoint in a form, so you can stop execution quickly when you get there. See "Break It Up Out There" below.

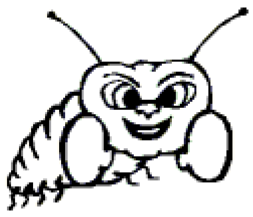
Once a form is open, though, the Object and Procedure drop-downs let you get to the various methods in the form and set breakpoints as needed (or you can do it with the Breakpoints dialog).

Just Watch Me!

Oh! death will find me long before I tire
Of watching you.
Rupert Brooke

The Watch window is the successor to the old Debug window. You put expressions in and you can see their values. But it does much more than that, too. You can set breakpoints (as you could in Debug) and you can change the values of variables, fields and properties on the fly.

Unlike the old Debug window, the Watch window lets you look at objects and arrays. Just put the object or array name in and it shows up with a "+" next to it to let you expand it and drill down. Very handy.



Watch out for ActiveX objects. Some of them don't interact so well with the Watch window and can hang VFP or make it play dead until you destroy the object. (Our tech editor also reports that he can crash VFP 5 under Windows 95 by putting a reference to a Word object in the Watch window.)

You can also drag and drop into the Watch window. If you want an expression that's almost the same as one that's already there, drag the one you have into the text box, edit it and hit Enter. You can drag from other windows as well, so you could drill down in the Locals window to find the property you want to watch, then drag it into the Watch window. When you do it that way, you don't even have to drop it in the text box first; just drop it right into the main part of the Watch window. You can do the same thing from the Trace window or even from VFP itself. No more typing long expressions and hoping you can get them right.

The Watch window lets you know what's changed recently. When the last command executed changed one of the items you're watching, the color in the Value column for that item changes. (Actually, it seems to stay changed for more than just the next command. We haven't figured out exactly when it changes back.)

To set a breakpoint in this window (as in the Trace window), click in the gray bar on the left. (As in the Trace window, a red dot appears next to the item with the breakpoint.) As soon as the value of that expression changes, program execution is suspended and you can get a look at the current state of affairs. (Note the difference between breakpoints in Trace, which happen before the marked line, and breakpoints in Watch, which happen after the value has changed.) With objects, you don't always know the name of the thing you want to look at. Say you want to see what's going on with a check box on a form. But the same form might be running several times. Instead of trying to figure out the name of the form, just refer to it via _Screen.ActiveForm—this expression always references the form that has the focus (unless it's a toolbar). Use _Screen.ActiveForm as a way of getting to the controls on the form. The nicest thing is that the Watch window is very forgiving—if the property or object you reference doesn't exist, no error is generated; you just have the message "(Expression could not be evaluated)" in the Value column for that expression. SYS(1270) is also a handy way to get a reference to an object so you can watch it. See that topic in the reference section for details.

Protected and Hidden properties are a special problem in the Watch window (and don't show up at all in the Locals window)—you can only see their values when you're in a method of the object. When you're executing other code (or even just sitting on a form), they show up as unable to be evaluated.

The columns in the Watch window can be resized. Just put the mouse over the divider between the columns. Voila, a sizer. We haven't yet found a good way to size things here so that we can always read everything that's showing, short of maximizing the debugger (and maybe the Watch window, too). But you can click into an item (click once on the line and again in the section you're interested in) and use the arrow, Home and End keys to see the hidden part.

Clicking into the Value section that way also lets you change the value, if it's possible. That is, you can change variable, field and property values, but you can't change the value of computed expressions. Very handy when you're 20 minutes into a complex test and you find you failed to initialize a counter to 0.

Local Hero

The evil which assails us is not in the localities we inhabit but in ourselves.
Seneca, *Moral Essays, "De Tranquillitate Animi" (On Tranquility of Mind)*

Just when you're convinced that the Watch window is the greatest thing since sliced bread, along comes the Locals window. This one cuts down dramatically on what you need to put into the Watch window. It shows you every variable that's in scope and, in fact, lets you choose the scope whose variables you want to see. Unlike the Watch window, you can't set breakpoints in Locals, but you can drill down in arrays and objects.

The context menu for this window gives you some control over which variables show up. The choices are a little strange, since they're not mutually exclusive. The Local and Public options do what they say—indicate whether variables declared as local and public, respectively, are shown. The Standard choice appears to really mean "private" and indicates whether private variables currently in scope are displayed, whether or not they were created by the current program. The Objects choice is independent of the other three and indicates whether variables holding object references are displayed, regardless of scope.

Call Me Anytime

She was not quite what you would call refined. She was not quite what you would call unrefined. She was the kind of person that keeps a parrot.
Mark Twain, *Pudd'nhead Wilson's New Calendar*, 1897

The Call Stack window lets you see where you are and where you've been. It shows the sequence of calls that got you to the current situation. However, it only shows call nesting—that is, if routine A calls routine B, which finishes, and then A calls C, which contains a breakpoint, the Call Stack window at the breakpoint shows only A and C. It doesn't show you that you visited B along the way.

Call Stack interacts with Locals and Trace. When you're stopped, you can click on any routine in the Call Stack window and the Locals window switches to show variables for that routine, while Trace shows the code for that routine.

At first glance, Call Stack isn't as useful as it should be, because it only works when Trace between breakpoints is on. Fortunately, though, as soon as you turn on Trace between breakpoints, the complete call stack does appear. Breakpoints also make the call stack appear. You can change the Trace between breakpoints setting with the context menu in the Trace window or the SET TRBETWEEN command, as well as in the Tools-Options dialog.

Here Comes Debug

We use the last of the Debugger windows a lot—it's the Debug Output window, and you can send all kinds of information there. By default, anything in a DebugOut command goes there, of course. The Event Tracker likes to send its output there, too.

Use the DebugOut command and the Debug Output window for the kinds of testing you've always done with WAIT WINDOWs or output sent to the screen. DebugOut interferes with your running program much less.

If you want to examine the output at your leisure, you can also redirect it to a file (either from Tools-Options or the SET DEBUGOUT command). If you forget to do so, the context menu for the window contains a Save As command. (You might want to think of it as a "save my bacon" command.)

Break It Up Out There

A prince never lacks legitimate reasons to break his promise.
Niccolò Machiavelli, *The Prince*, 1514

Breakpoints are one of the key weapons in the fight to make code work right. They let you stop where you want to see what's going on. As with so much else about debugging, breakpoints got much better in VFP 5.

In the old debugger, you could set breakpoints at a particular line of code or when a specified expression changed. With some creativity, you could stop pretty much anywhere, but it wasn't easy.

The new debugger makes it much simpler. First of all, you can set breakpoints while you're looking at the code in the development environment. The context menu for all the code editing windows includes a Set Breakpoint option.

Once you're working with the debugger, the breakpoint dialog is available from the Tools menu and has its own button on the Debugger toolbar. In addition to the same old choices, it also has options to stop at a specified line when a particular expression is true or after you've executed it a certain number of times. You can stop not only when an expression has changed, but also when an expression is true. Each breakpoint you specify can be turned on and off independently. Finally, breakpoints are among the things saved when you save the debugger configuration.

In an OOP world, it can be difficult to specify just where you want to put a breakpoint. What a pain to type in "_Screen.ActiveForm.grdMain.Columns[3].Text1.Valid". Fortunately, you don't have to. If you don't mind stopping at every Valid routine that contains code, just specify Valid. If you do mind stopping at all of them, maybe you don't mind stopping at each Valid in that form—specify Valid for the Location and the form's name (including the SCX extension) for the File. And so forth. And so on. Be creative and you'll get just what you want.

Some things are obvious candidates for breakpoints. If a variable is coming out with a value you don't understand, set a "Break when expression has changed" breakpoint on it. Similarly, if a setting is being changed and you can't figure out where, set the same kind of breakpoint on SET("whatever").

You can get even more clever than that. Wanna find out when a table is being closed? Set a breakpoint on USED("the alias"). When is a window being defined? Set a breakpoint on WEXIST("the window name").

You get the idea—you can set breakpoints on any change at all in program state as long as there's some way to express it in the language.

In fact, you can take advantage of this flexibility to set breakpoints in the Watch window as well. Put an expression like "CLICK"$PROGRAM() in the Watch window and set a breakpoint (by clicking in the left margin next to the item) to make VFP stop as soon as it reaches any Click method. You can use the same approach to set breakpoints on things like the line number executing. While the Breakpoint dialog gives you tremendously flexibility in describing your breakpoints, turning them on and off is much easier in the Watch window.

Boy, What an Event That Was

Men nearly always follow the tracks made by others and proceed in their affairs by imitation, even though they cannot entirely keep to the tracks of others or emulate the prowess of their models.
Niccolò Machiavelli, *The Prince*, 1514

One of the trickier aspects of working in an OOP language is that events just happen. For programmers used to controlling every aspect of an application's behavior, this can be disconcerting to say the least. Event Tracking is one way to calm your rapid breathing and get your blood pressure under control.

Like the breakpoint dialog, event tracking lives in the debugger's Tools menu and on the Debugger toolbar. It lets you choose whichever events you're interested in and have a message appear whenever that event fires (for any object at all). The messages go into the Debug Output window by default, but can also be sent to a file.

Be careful. Tracking an event like, say, MouseMove can generate a tremendous amount of output. Choose only the events you're really interested in, so the output is manageable.

One thing about event tracking is one of our least favorite aspects of the debugger. The button for it on the Debugger toolbar is a toggle. In fact, it's not really a button at all, it's a graphical check box. So, when you have event tracking in place, clicking the "button" doesn't bring up the dialog for you to change it—it turns it off. Then, when you click again to make changes, the check box in the dialog that actually controls tracking is unchecked. Yuck. This wouldn't be such an annoying problem except that there's no menu shortcut for event tracking, either. So you can't just hit a key combo to bring up the dialog. We sure hope someone at Microsoft notices how aggravating all this is soon. (We've told them.)

Cover Me, Will You?

We gaze up at the same stars, the sky covers us all, the same universe compasses us. What does it matter what practical systems we adopt in our search for the truth. Not by one avenue only can we arrive at so tremendous a secret.
Quintus Aurelius Symmachus, *Letter to the Christian Emperor Valentinian II*, 384

The last of the debugger's tools is Coverage Logging. This gadget creates a file (generally, voluminous) containing one line for each line of code that executes. It includes the filename, the routine name, the line number, the object containing the code, how long it took, and the nesting depth of the routine.

The file is comma-delimited, so it's easy to pull into a table. However, by itself, the file isn't terribly informative. Unfortunately, in VFP 5, that's all we had.

VFP 6 includes the Coverage Profiler, an application that analyzes coverage logs and gives you information such as how many times a given line was executed. If you point it to the right project, it'll tell you which files in that project were called and which ones weren't.

The Coverage Profiler that's provided is actually just a front end on an incredibly extensible coverage engine. You can enhance it in two different ways. The first is by specifying Add-Ins, similar to those you can specify for the Class Browser. The second alternative is to subclass the coverage class and define your own front end for it. We haven't had a chance yet to explore either of these, but we suspect there are a lot of possibilities.

The Coverage button on the Debugger toolbar has the same annoying behavior as the Event Tracking button, but we find it bothers us a lot less because we're far less likely to need to make changes and keep going. Also, all it takes to turn on Coverage Logging is a valid filename—there's no check box you have to remember to check.

Doing it the Old-Fashioned Way

You've set breakpoints, you've put everything you can think of in the Watch window, you tracked events until they're coming out of your ears, and you still can't figure out where it's going wrong. Time to step back and try another approach.

Grab a sheet of paper and a copy of the code (could be online or on paper). Now pretend you're the computer, and execute your program step by step. Use the paper to keep track of the current values of variables.

These days, we don't use this technique a lot, but when we do, it's invaluable. This systematic attack, or any method of logically proceeding through the possibilities, whether splitting the problem in half or bracketing it from input to output, is vastly superior to the panicked "change this, change that, change the other thing, try it again" mentality typical of amateurs. If you can't explain why it works, you haven't found the problem yet.

Staying Out of Trouble

Nobody knows the trouble I've seen.
Anonymous Spiritual

These are techniques we use to minimize our troubles. They're mostly pretty straightforward, once you know about them.

Get Me Outta Here

Always have an escape valve. Sometimes programs crash really badly and leave you with lots of redefined keys and all kinds of other trouble. Keep a cleanup program around, which at a minimum should include:

```
ON ERROR
ON KEY
SET SYSMENU TO DEFAULT
CLOSE ALL
CLEAR ALL
```

Give it a simple name like Kill.PRG and set some obscure key combination (like CTRL+Shift+F12) to DO this program. If your applications use an application object of some sort, put all the cleanup code in the app object's Destroy method and you'll cut way down on the number of times you need the escape hatch, since Destroy should run even if the application crashes.

Step By Step

Testing a whole application all at once is guaranteed to fail. Build it piece by piece and test each piece as you go. It's much easier to get a 20-line function working than a 20,000-line application. But if the 20,000-line app is made up of 20-line functions that work, it's a whole lot easier.

When you're testing, take it one thing at a time, too. For that 20-line function, see what happens if you forget the parameters, if you pass the wrong parameters, if you pass them in the wrong order. Once all that stuff is working, get to the heart of the thing and see if it works when you do hand it the right data. Try each endpoint; try typical data; try bizarre, but legal, data.

When building a complicated routine (especially parsing sorts of things), we've even been known to test one line at a time from the Command Window until we get it right, then plunk it into the routine. The added-in-VFP 5 ability to highlight several lines of code in the Command Window or a MODI COMM window and execute them makes this technique more valuable than ever.

You Deserve a Break

When your hair is all gone and you're pounding your bald head against the wall and you still can't see what's wrong, take a break. Go for a walk, talk to a friend, anything—just get away from your desk and let the other part of your brain kick in. The folk wisdom of "sleeping on it" really works for many difficult troubleshooting problems.

That's What Friends Are For

If a break doesn't work, ask somebody else. We've both worked alone (Tamar still does), but we're not afraid to pick up the phone and call a friend to say, "What am I doing wrong here?"

If we don't need an instant answer, we'll post a message in the appropriate Fox forum on CompuServe—we've never failed to get some ideas there, even if no one knows the exact answer. (See "Back 'o Da Book" for some other places to get VFP help online.)

So What's Left?

When you have eliminated the impossible, whatever remains,
however improbable, must be the truth.
Sir Arthur Conan Doyle, *The Sign of Four*, 1890

So, you're sure the file exists or that the variable is getting properly initialized. If you've tried everything else and you can't find the problem, question your assumptions. Go up a level and make sure things are as you expect when you get to the problem point. Sherlock Holmes knew what he was talking about.

Do It Right in the First Place

A lot of the things we mention elsewhere in this book will help to keep you from reaching the hair-pulling stage. It's a lot easier to realize you've initialized a variable incorrectly when you see something like:

```
cItem = 7
```

than when it's:

```
MyChosenItem = 7
```

Good documentation makes a difference, too. If you've declared cItem as LOCAL and commented that it contains "the name of the chosen item," you're not likely to make the mistake in the first place.

These are a Few of our Favorite Bugs

Nothing is more damaging to a new truth than an old error.
Johann Wolfgang von Goethe, *Sprüche in Prosa*

There are certain bugs we find ourselves fixing over and over and over again. Since we make these mistakes a lot, we figure other people do, too. (Check out "It's a Feature, Not a Bug" for more items along these lines.)

Is That "a AND NOT b" or "b AND NOT a"?

It's easy to mess up complicated logical expressions. Back to paper and pencil to get this one right. Make what logicians call a "truth table"—one column for each variable and one for the final result. Try all the possible combinations of .T. and .F. and see if you've got the right expression.

If the expression's really complex, break it into several pieces and check each one separately.

But I Changed That Already

Change begets change. Nothing propagates so fast.
Charles Dickens, *Martin Chuzzlewit*, 1844

You find a bug. You figure it out and fix the code. You run it again and nothing's different—the exact same problem you will swear you just fixed replays again and again. What gives? There are a couple of possibilities here.

One is that you're running an APP, and though you changed an individual routine, you didn't rebuild the APP file. Do.

Another is that you're not running the copy of the program you think you are. Check around for other copies in other directories and figure out which one you're actually running. Try deleting the FXP file, if you're dealing with a PRG. If you don't get a new FXP, you'll know you're not pointing at the program you think you are.

There's one more subtle case that can happen. FoxPro keeps stuff in memory to speed things up. In some situations, it doesn't do a good job of cleaning up when you need it to. If all else fails, try issuing CLEAR PROGRAM before you run again. Really desperate? QUIT and restart.

What is This?

In moving to object-oriented programming, we often forget to include the full reference needed to talk to an object. Writing FOR nCnt = 1 TO ControlCount doesn't do a bit of good—it needs to be This.ControlCount or ThisForm.ControlCount or This.Parent.ControlCount or something like that. We've found we're most likely to forget when dealing with array properties.

There are no easy solutions for this one—we all just have to learn to do it right.

I Really Do Value Your Input

Here's another one that you just have to learn, but it's pretty common for VFP newcomers and we still do it ourselves occasionally, especially when working with unfamiliar objects. The problem is referring to the object when you want one of its properties. Probably, this happens most with the Value property. Somehow, it seems appropriate that ThisForm.txtCity should contain the city that the user just typed in, but of course, it doesn't. You need ThisForm.txtCity.Value in this case.

S3C4. Faster Than a Speeding Bullet.

Faster Than a Speeding Bullet

Speed is where it's at. No client is going to pay you to make his app run slower. Fox Software's initial claim to fame was that FoxBASE (and later, FoxBASE+ and FoxPro) ran faster than its competitors. Fox's reputation for speed was well deserved. However, the speed gain has never been automatic. You have to do things right to make your code "run like the Fox."

The Rushmore optimization technology introduced in FoxPro 2.0 is based on indexes. Rushmore examines index files to determine which records meet the conditions for a particular command. So, in order to make things fast, it's important to create the right indexes and to write code that takes advantage of those indexes. The difference between the lightning-fast code your client uses to make crucial strategic decisions and the plodding code his competition uses might differ by no more than an operator or two, so listen up!

There are also some other tricks, not related to Rushmore, that can speed up your applications considerably. This section discusses both Rushmore and non-Rushmore optimization techniques.

Scared by a Mountain in South Dakota?

Fox Software always claimed that Rushmore was named after Dr. Fulton and the development team watched Hitchcock's *North by Northwest*. But we have no doubt the name caught on due to the phrase "rush" embedded in it. In fact, some of the FoxPro documentation and advertising used the phrase "RUSH me some MORE records."

As we mentioned above, the key to getting the most from Rushmore is to create the right indexes and take advantage of them. So how do you know which are the right indexes, and how do you take advantage of them?

Rushmore can optimize the SET FILTER command, and any command involving a FOR clause, as well as SELECT-SQL. The secret (not really a secret—it is documented) is to make the left-hand side of each expression in the filter, FOR or WHERE clause exactly match an existing index tag. For example, to optimize:

```
SUM OrderTotal FOR state="PA"
```

you need an index tag for state. If your tag is on UPPER(state), instead, you'd want to write the command as:

```
SUM OrderTotal FOR UPPER(state)="PA"
```

Suppose you want to find everyone named Miller in a table of Clients and that you have a tag on UPPER(cLastName+cFirstName) to put folks in alphabetical order. You optimize the BROWSE by writing it as:

```
BROWSE FOR UPPER(cLastName+cFirstName)="MILLER"
```

even though you're really interested only in the last name.

It's All in What You Index

We've answered the second question—how to take advantage of existing tags—but we still haven't tackled the first: What are the right indexes to create? That's because it's not always straightforward. There are a few clear-cut rules, but to a great extent, you'll need to use your judgment and test your theories against your data, on your hardware. Here are the rules:

- Create a tag for your primary key, the field that uniquely identifies the record. (Do this whether or not you define it as a primary key in the database.) You'll need it to look up particular records and for setting relations. (If your table is in a database, you'll want a tag for the primary key anyway for creating persistent relations.)

- Create a tag for any field or expression you expect to search on frequently.
- Create a tag for any field or expression you think you'll want to filter data on frequently. (This is to let Rushmore kick in.)
- Make sure the tags you create exactly match the conditions you'll need to search or filter on.

- Don't automatically create tags on every field. (That's called inverting the table.) It can make adding and updating records slower than necessary, especially if you have a lot of fields in your table. On the flip side, if you have a table, especially one that is rarely changed, and you do use every field in filters, then go ahead and invert the table.
- Do not create indexes with a NOT expression for Rushmore optimization. Rushmore ignores any tag whose expression contains NOT. If you need the NOT expression, say, for a filter, create both indexes, one with and one without the NOT.
- Don't filter your tags. That is, don't use the FOR clause of the INDEX command. Tags that are filtered are *ignored* by Rushmore. If you need a filtered tag for some reason, and you're likely to filter on that tag's index expression as well, create an unfiltered tag, too.

In general, you'll be trading off update speed for search speed. So, think about what you expect to do with this table. If it's going to have lots of additions but few searches, keep the number of tags to a minimum. If it'll be used for lots of searching, but rarely updated, create more tags.

But I Didn't Delete Any Records!

One of the optimization tips that fools lots of people has to do with SET DELETED. The typical conversation goes something like this:

"I have a query that's taking too long. How can I speed it up?"

"Create a tag on DELETED() for each table, if you have SET DELETED ON."

"But there are only a few deleted records. That shouldn't make much difference."

"Try it anyway."

(Later)

"You're right. It's much faster now. But there are only a few deleted records. How come it matters so much?"

What's going on here? In fact, you'll see the same speed-up even with NO deleted records. It's the setting of DELETED that matters.

Here's the point. Even in many complex queries and FOR clauses, Rushmore performs its magic almost entirely on the relatively small and compact CDX file, a file structured with nodes, branches and leaves to be searched efficiently. When DELETED is ON, FoxPro has to check each and every record in a result set (whether from a query, a filter, or FOR) to see if it's deleted—even if no records are actually deleted. This sequential reading of the entire cursor or file completely defeats the benefits of Rushmore. Don't do it!

By creating a tag on DELETED(), you let Rushmore do the checking instead of looking at each record sequentially, which makes the whole thing much faster. The larger the result set, the more speed-up you'll see.

Going Nowhere Fast

Another common problem goes like this. In troubleshooting sessions we attend, someone complains that a filter should be optimized, but it's dog slow. He's asked to show the filter and the tags. Everything looks good for Rushmore to optimize the filter. Puzzling.

Then he shows the code he's using. Typically, it looks something like this:

```
SET FILTER TO <something optimizable>
GO TOP    && put filter in effect
```

and the light goes on. GO TOP and GO BOTTOM are not optimizable commands. They move through the records sequentially, attempting to find the first record matching the filter.

Without a filter (and with SET DELETED OFF), this isn't generally a problem. Moving to the top or bottom of the current order is pretty quick. FoxPro can either locate the first or last record in the index or, if no tag is set, move directly to the beginning or end of the file.

But when a filter is set (or DELETED is ON, which is like having a filter set), once GO gets to the first or last record in the order, it has to search sequentially for the first record that matches the filter condition. This is what's so slow. Smart like a fox, eh? What a dumb idea! This is like you writing code to go to record 10 by issuing a SKIP, asking if this is RECNO()=10, and if not, SKIPping again.

What can you do about it? Don't use GO TOP and GO BOTTOM. How do you avoid them? By using a neat trick. It turns out that LOCATE with no FOR clause goes to the first record in the current order. So, for GO TOP, you just issue LOCATE, like this:

```
SET FILTER TO <optimizable condition>
LOCATE   && same as GO TOP
```

Okay, that works for finding the first record. What about the last record? You have to stand on your head for this. Well, almost. You really have to stand the table on its head. Try it like this:

```
SET FILTER TO <optimizable condition>

* reverse index order
lDescending=DESCENDING()
IF lDescending
   SET ORDER TO ORDER() ASCENDING
ELSE
   SET ORDER TO ORDER() DESCENDING
ENDIF
* now Top is Bottom and Bottom is Top
LOCATE  && same as GO TOP

IF lDescending
   SET ORDER TO ORDER() DESCENDING
ELSE
   SET ORDER TO ORDER() ASCENDING
ENDIF
```

After setting the filter (or with a filter already in effect), you turn the index upside down. If it was ascending, you make it descending; if it was descending, you make it ascending. Then, use LOCATE to go to the first record. Since you've reversed the order, that's the last record in the order you want. Then, reverse the order again. Voila! You're on the bottom record.

By the way, the code above works only if there is an index order set. If there might be no order, you have to check for that.

One more warning. Under particular circumstances, the work-around can be very slightly slower than just using GO. In most cases, though, it tends to be an order of magnitude faster. We think it's worth it.

HAVING noWHERE Else To Go

SQL-SELECT has two clauses that filter data: WHERE and HAVING. A good grasp of the English language might lead us to believe that these are synonyms, but SQL is not English, and mixing these two indiscriminately is a sure-fire disaster in the making! It's not obvious where a particular condition should go at first glance. But getting it wrong can lead to a significant slowdown.

Here's why. The conditions in WHERE filter the original data. Wherever possible, existing index tags are used to speed things up. This produces an intermediate set of results. HAVING operates on the intermediate results, with no tags in sight. So, by definition, HAVING is slower than WHERE, if a query is otherwise constructed to be optimized.

So, when should you use HAVING? When you group data with GROUP BY and want to filter not on data from the original tables, but on "aggregate data" formed as the results of the grouping. For example, if you group customers by state, counting the number in each, and you're interested only in states with three or more customers, you'd put the condition COUNT(*)>=3 in the HAVING clause.

```
SELECT cState,COUNT(*) ;
       FROM Customer ;
       GROUP BY cState ;
       HAVING COUNT(*)>=3
```

A simple rule of thumb: Don't use HAVING unless you also have a GROUP BY. That doesn't cover all the cases, but it eliminates many mistakes. To make the rule complete, remember that a condition in HAVING should contain one of the aggregate functions (COUNT, SUM, AVG, MAX or MIN) or a field that was named with AS and uses an aggregate function.

The Only Good Header is No Header

FoxPro lets you store procedures and functions in a variety of places. But using the Project Manager gives you a strong incentive to put each routine in a separate PRG file. We generally agree with this choice.

But, if you're not careful, there's a nasty performance penalty for doing so. It turns out that having a PROCEDURE or FUNCTION statement at the beginning of a stand-alone PRG file increases the execution time by a factor of as much as 10!

You read that right. It can take 10 times as long to execute a PRG that begins with PROCEDURE or FUNCTION as one with no header. Hearing about this goodie (no, we didn't discover it ourselves), we tested a couple of other alternatives. It turns out that using DO <routine> IN <PRG file> cuts the penalty down some, but it's still twice as slow as simply eliminating or commenting out the header line.

SETting PROCEDURE TO the PRG, then calling the routine, speeds things up if you only have to do it once, but issuing SET PROCEDURE TO over and over again (as you'd need to for many different PRGs) is about 20 times slower than the slow way. That is, it's 200 times slower than omitting the header in the first place.

But wait, there's more. Not surprisingly, if the routine you're calling isn't in the current directory, but somewhere along a path you've set, it takes a little longer. For an ordinary routine with no header, the difference isn't much. Same thing if you're using SET PROCEDURE (which you shouldn't be, except for coded class libraries). However, the other two cases get a lot slower when they have to search a path. Using DO <routine> IN <PRG file> when the file isn't in the current directory is just about as slow as doing a SET PROCEDURE. But that's only the bad case. The horrible situation is calling a routine with a PROCEDURE or FUNCTION header directly—it can be as much as 1000 times slower than calling the same routine without the header!

The good news is that the path penalties go away as soon as you add the routines to a project and build an APP or EXE. That is, unless you're running in a very unusual setup, your users are unlikely to pay this price.

Bottom line. When you migrate a routine into a stand-alone PRG file, comment out the header line and just start with the code. It's easy and it'll speed up your applications considerably.

Loops Aren't Just for Belts

FoxPro offers three different ways to write a loop. Choosing the right one can make a big difference in your program. So can making sure you put only what you have to inside the loop.

Let's start with the second statement. Every command or function you put inside a loop gets executed every time through the loop. (Big surprise.) Put enough extra stuff in there and you can really slow a program down. The trick is to put each statement only where you need it. This is especially true when you've got nested loops—putting a command farther in than it has to be might mean it gets executed dozens more times than necessary.

Bottom line here: If the command doesn't depend on some characteristic of the loop (like the loop counter or the current record) and it doesn't change a variable that's changed elsewhere in the loop, it can probably go outside the loop.

Here's an example:

```
* Assume aRay is a 2-D array containing all numeric data
* We're looking for a row where the sum of the first three columns is
* greater than 100
lFound = .F.
nRowCnt = 1
DO WHILE NOT lFound AND nRowCnt<=ALEN(aRay,1)
```

```
   IF aRay[nRowCnt,1]+aRay[nRowCnt,2]+aRay[nRowCnt,3]>100
      lFound = .T.
   ELSE
      lFound = .F.
      nRowCnt=nRowCnt+1
   ENDIF
ENDDO
```

The version below eliminates repeated calls to ALEN() and the need for the lFound variable. Benchmarks with 10,000 records show that it's almost twice as fast as the original.

```
nNumofRows = ALEN(aRay,1)
DO WHILE aRay[nRowCnt,1]+aRay[nRowCnt,2]+aRay[nRowCnt,3] <= 100 and ;
         nRowCnt < nNumofRows
  nRowCnt = nRowCnt + 1
ENDDO
```

We find we're most likely to make this particular mistake when we're dealing with nested loops, so scrutinize those especially.

What's This Good FOR?

In the case of loops that execute a fixed number of times, FOR is a better choice than DO WHILE. Because the counting and checking feature is built into FOR, it just plain goes faster than DO WHILE. In a simple test with a loop that did nothing at all except loop, FOR was more than 10 times faster than DO WHILE. Never write a loop like this:

```
nCnt = 1
DO WHILE nCnt <= nTopValue
   * do something here
   nCnt=nCnt+1
ENDDO
```

Always use this instead:

```
FOR nCnt = 1 TO nTopValue
   * do something here
ENDFOR
```

SCANning the Territory

Guess what? DO WHILE isn't the best choice for looping through records either. SCAN was designed to process a table efficiently and does it faster than DO WHILE. Our results show that SCAN is one-and-a-half to two times faster to simply go through an unordered table one record at a time. (This is where we have to come clean and admit that the phenomenal differences we reported in the original *Hacker's Guide* appear to have been flawed. We're seeing about the same results in VFP 3.0b and 5.0a as we are in 6.0.)

To give full disclosure, we did find that with some index orders, DO WHILE was as much as 20 percent faster. With other indexes, SCAN is faster, although it doesn't appear to have the same advantage as in an unordered table. (It's also worth noting that, with large tables, if the memory allocation to FoxPro isn't property tuned—see below—DO WHILE can be faster than SCAN.)

A word to the wise here: Don't just globally replace your DO WHILE loops with SCAN...ENDSCAN. SCAN has a built-in SKIP function—if your code already has logic to perform a SKIP within the loop, you can inadvertently skip over some records. Make sure to pull out those SKIPs.

To Wrap or Not to Wrap

One of the new capabilities that OOP gives us is "wrapper classes." These classes let us take a collection of related capabilities and put them all into a single class. The class gives us a more consistent interface to the functions involved and generally presents a tidy package.

The Connection Manager class described in the Reference section (see SQLConnect()) is pretty much a wrapper class, though it adds some capabilities. We've seen folks suggest wrapper classes for the FoxTools library (which desperately needs a consistent interface despite the addition of lots of its residents to the language). During the beta test for VFP 3, we played around on and off for months with a wrapper class for array functions that would let us stop worrying about things like the second parameter to ALEN().

On the whole, wrapper classes sound pretty attractive. Unfortunately, they also add a fair amount of overhead.

There's another way to do the same thing—just create an old-fashioned procedure file. Now that SET PROCEDURE has an ADDITIVE clause, it's no big deal to have lots of procedure libraries around. It turns out, of course, that procedure libraries also carry an overhead penalty.

Because the contents of the class or library matter so much, it's hard to produce benchmarks that give you hard and fast rules about this stuff. We tested with our embryonic array handler class, using only some of the simpler methods included (aIsArray, aElemCount, aRowCount, aColCount, aIs2D—all of which do exactly what their names suggest). We set it up as a class and as a procedure library. Then, we wrote a program that made a sample series of calls. We also wrote the same functionality in native code (ALEN() for aElemCount, ALEN(,1) for aRowCount and so on).

The sad result is that either a procedure library or a class is an order of magnitude slower than using the built-in functionality. In this example, the procedure library is faster than the class by about a third.

We also tested the same functions as stand-alone programs. The timing came out pretty much the same as the procedure library and the class. (The difference between this case and the timing reported in "The Only Good Header is No Header" is that a single SET PROCEDURE was used in this case rather than issuing SET PROCEDURE for each function call.)

Finally, we tested with everything (the test program, the stand-alone programs, the procedure file and the class definition) built into a single APP file. Using an APP improved the speed of each case a little, but didn't make a significant difference overall.

Our guess is that, as functionality becomes more complex, the overhead counts less. Given the other, overwhelming benefits of using modular code, we don't recommend you stop writing procedures. But, at this point, we can't recommend wrapper classes where a procedure library would do.

There are some benefits to a wrapper class, of course. The biggest benefit is the ability to subclass to provide specialized behaviors. Where this is a possibility, it's worth the overhead.

What's in a Name?

You wouldn't think that a little thing like a name would matter so much. But it does. The name we're referring to is the Name property possessed by almost every object you can create in Visual FoxPro. (A few of the weird marriages of Xbase to OOP, like SCATTER NAME, produce objects without a Name property.)

When you CreateObject() an object whose class definition doesn't assign a value to the Name property, Visual FoxPro makes one up for you. That's nice. Except it insists on making it unique (usually, the class name or a variant thereof, followed by one or more digits, like Form3 or Text17). The problem is, as the number of objects of that class grows, making sure a name is unique takes longer and longer. The Microsoft folks say the time grows exponentially. We suspect that's an overstatement and that it's actually geometric. What it ain't is linear. What it really ain't is fast enough. (Before we go any further with this, we should point out that this applies only to code classes. All VCX-based classes have an implicit assignment of the Name property.)

We tested with a pair of very simple classes based on Custom. One contained nothing. The other contained an explicit assignment to Name. With 10 repetitions, the explicitly named class would instantiate so fast it couldn't be measured, but the nameless class was fast, too. By 100 repetitions, explicit naming was more than four times as fast. At 1,000 repetitions, the explicit version was eight to 10 times faster. At 5,000 of each class, explicit names are about 18 times faster than nameless objects to instantiate.

The moral of the story here is easy. Always assign a value to the Name property for any class you write in code.

Incidentally, it turns out that getting rid of all these objects once you instantiate is pretty expensive, too. Working with VCX-based classes, it took almost six times as long to destroy 5000 of the same object than to create them. With our explicitly named code class, destroying 5000 instances took two to three times as long as creating them.

Looks Can Be Deceiving

But, in this case, they're not. The form property LockScreen lets you make a series of changes to a form without the individual changes showing as you go. When you set LockScreen to .F., all the changes occur simultaneously. Visually, it's far more consistent.

We were all set to tell you that this is one of those times where the user's eyes will play tricks on him. He'll think the update is faster because he doesn't see the individual changes take place.

But guess what? The update really is faster this way. We tested a simple form with just a few controls. We changed only a few properties of the form (Height, Width, BackColor, ForeColor and Caption) once each. With LockScreen set to .T., the updates were about one-and-a half times faster. Surprise—the version that looks better is faster, too. We suspect it's because Windows only has to redraw the screen once.

What Type of Var are You?

Testing the type of a variable or field is one of those things we do a lot in our code. In VFP 6, it's something we can do faster than ever. The new VARTYPE() function is significantly faster than its predecessor, TYPE(). How much faster? With both variables and fields, we consistently find VARTYPE() three to four times as fast as TYPE().

One warning here. VARTYPE() is appropriate only for fields, variables, properties and the like. You can't use it with expressions to find out what type the result will be. In that case, you need TYPE(), which pseudo-evaluates the expression to find out the result type. VARTYPE() simply looks at what you pass it and tells you its type. So, VARTYPE("x+y") returns "C", regardless of the type of x and y, while TYPE("x+y") returns "N". So, don't throw TYPE() out of your toolkit quite yet.

How to Create an Object and Other Mysteries of Life

VFP 6 also introduces a new way to create objects. The NewObject() function lets you instantiate objects without worrying about whether you've pointed to the class library ahead of time—instead, you just include the library name in the call. CreateObject(), of course, needs a Set ClassLib or SET PROCEDURE ahead of time.

So which way is faster? As usual, the answer is "it depends." With VCX-based classes, if you can issue Set ClassLib just once and then instantiate classes from that library repeatedly, CreateObject() is the way to go. It's anywhere from one-and-a-half to eight times faster than calling NewObject() with the class library. On the other hand, if you need to load the library each time, the Set ClassLib/CreateObject() pair is more than an order of magnitude slower than NewObject().

How about for classes written in code? In that case, issuing a single SET PROCEDURE and calling CreateObject() repeatedly is an order of magnitude faster than either NewObject() or the SET PROCEDURE/CreateObject() pair, which are pretty similar.

Oh, and one more—instantiating a coded class is a little faster than instantiating a VCX-based class. However, except in the case where you're setting the library each time (which you should never do), it's not enough faster to wipe out the benefits of developing classes visually.

We tested and found no performance penalty for having a lot of class libraries open, no matter where in the list the class you're instantiating is found. So the rule here is to think about how you're going to do things before you write the code and, if possible, just keep open the class libraries you use a lot. Then use NewObject() for the one-shots, the classes from libraries you need only once in a while.

Can You Have Too Much Memory?

It turns out that, in VFP, the answer is "yes." When you start VFP, it figures out how much memory it ought to be able to use, if it needs it. The number is generally about half as much as the machine actually has. Often, the amount that VFP picks is too much.

How can you have too much memory? Like this, according to our buddy Mac Rubel, who knows more about this topic than anyone else—more even, we suspect, than the folks who wrote VFP. However much memory VFP grabs, it assumes it has that much *physical* memory to work with. But, because it takes so much memory, it often doesn't—some of the memory it's working with is really disk space pretending to be memory, and that's slow. By decreasing the amount of memory VFP thinks it has available, you ensure that it only uses physical memory. VFP knows what to do when it needs more memory than it has available, and it's good at that. The last thing you want happening is the operating system swapping virtual (disk) memory for real memory while FoxPro thinks it is using RAM. So, as long as you restrict it to using physical memory, things are fast, fast, fast.

Okay, so how do you that? Use the SYS(3050) function. SYS(3050,1) controls foreground memory, the memory VFP has available when it's in charge. SYS(3050,2) is for background memory—how much memory FoxPro should have when you're off doing something else. In either case, you pass it a number and it rounds that down to a number it likes (multiples of 256) and that's how much memory it uses. It even tells you how much it really took.

We were really amazed how much of a difference this setting makes. On Tamar's machine with 64MB of RAM, VFP 6 takes 35,072 MB by default. Reducing it to just under 24,000 MB (by calling SYS(3050,1,24000000)) cut the time needed for one of the particularly slow tests in this section by a factor of 4! One disclaimer here: Tamar tends to operate with lots of applications open. A typical load while working on this book was six or seven apps running (not to mention those living in the system tray, like Dial-Up Networking and a virus checker). No doubt they all take some memory.

Practice, Practice, Practice

All of the tips we've given you here should speed up your code, but *your* application on *your* LAN with *your* data is the true test. Differences in network throughput, the architecture of your system, the design of your tables, your choice of indexes, the phase of the moon, what's on TV that night, and so forth all make significant differences in the performance you see. Our advice is always to examine and benchmark how a particular change affects your system. Keep in mind that a single test isn't conclusive unless it can be repeated, and you need to repeat tests with caution because FoxPro and the operating system and the network and even your disk controller might be caching information.